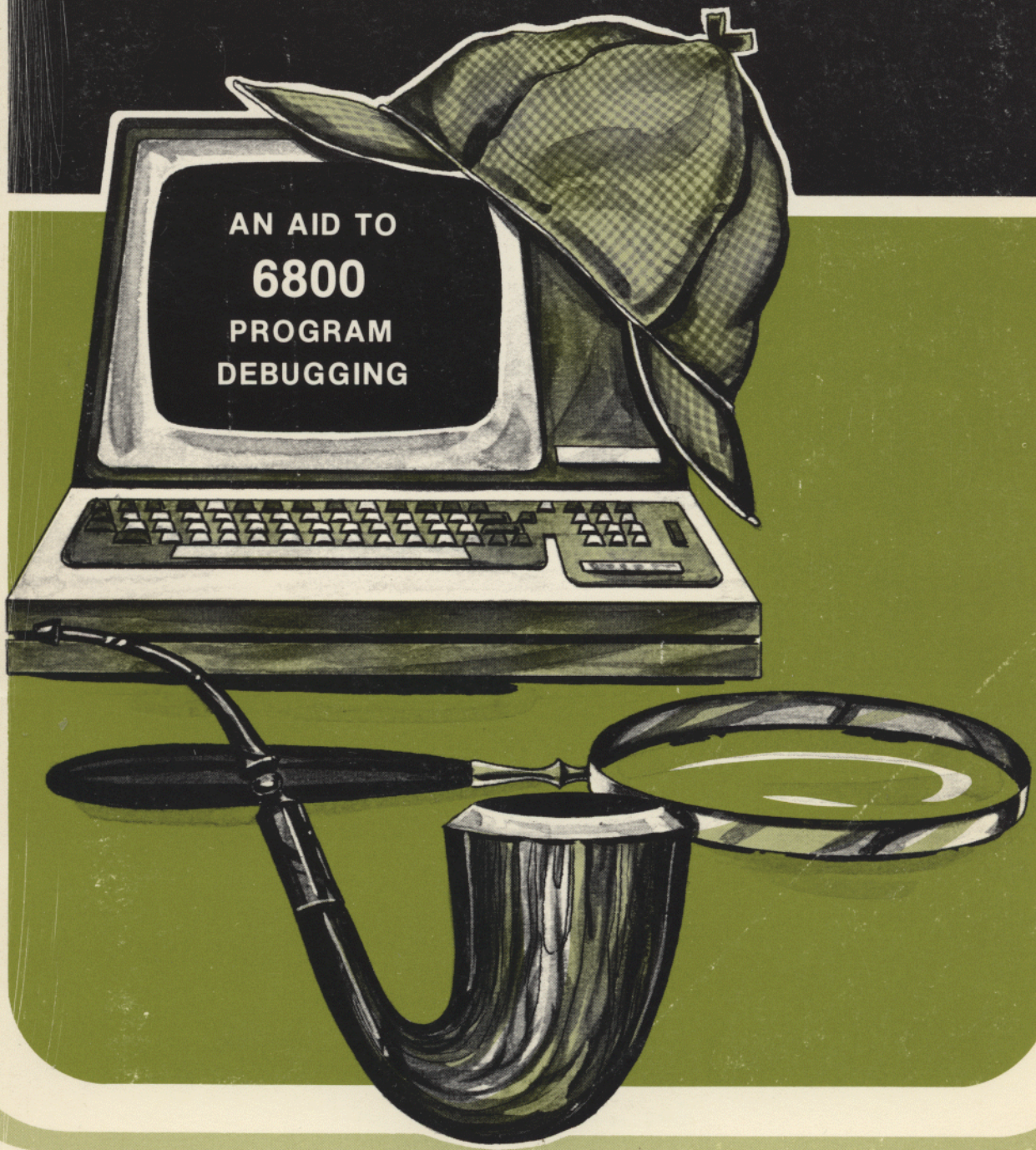


\$6.00
A PAPERBYTETM BOOK

6800
TRACER
GRAPPEL/HEMENWAY



Tracer

A 6800 Debugging Program

by
Robert D Grappel and **Jack E Hemenway**

BYTE Publications, Inc.
70 Main Street
Peterborough, New Hampshire 03458

Copyright © 1978 BYTE Publications
Inc. All Rights Reserved. BYTE and
PAPERBYTE are Trademarks of BYTE
Publications Inc. No part of this book
may be translated or reproduced in any
form without prior written consent from
BYTE Publications Inc.

The program Tracer is copyright © 1977 Robert D. Grappel and Jack E.
Hemenway, and is reproduced with the authors' permission under exclu-
sive book publishing license.

Library of Congress Cataloging in Publication Data

Grappel, Robert D

Tracer : a 6800 debugging program.

1. Motorola 6800 (Computer)--Programming.
 2. Tracer (Computer program). I. Hemenway, Jack E.,
joint author. II. Title.
- QA76.8.M67G73 001.6'425 78-7326
ISBN 0-931718-02-3

Printed in the United States of America

Table of Contents

Jack and the Machine Debug	5
Sample Tracer Output (Tracing Tracer)	11
Tracer Program Notes	12
Listing 1: Tracer Assembly and Source Listing	14
Table 1: Sorted Symbol Table for the Above Assembly	22
Table 2: Table of Hexadecimal Data for the Character Strings of Listing 1	22
Tracer Object Code Listing	23
Machine Readable Object Code (Bar Codes)	24

*Cover Design: Dawson Advertising Agency, Inc.
Concord, New Hampshire*

Artist: George Lallas

Dear Reader,

It has come to our attention that a bug appears in our book, TRACER: AN AID TO 6800 PROGRAM DEBUGGING.

As written, the Change SP function only changes the data stored at \$0070. When you Continue Trace, that data is thrown away and is replaced by the current Stack Pointer address, so nothing is changed. By making the following modification to the Tracer program, everything works as intended and stack reference data is retained in the new stack location to a level of 16 bytes:

				* CHANGE TO		
020B	7E	030B	R	JMP SCONT		continue change stack pointer
				* ADD		
030B	FF	0070	R	SCONT	STX STACKP	store it
030E	09				DEX	
030F	09				DEX	
0310	5F				CLRB	zero counter
0311	08			OVR	INX	line up new & old stacks
0312	32				PULA	data from old stack
0313	A7	00			STAA O,X	store it in new stack
0315	5C				INCB	count
0316	C1	10			CMPB \$10	done transfer data ?
0318	26	F7			BNE OVR	
031A	8E	0070	R		LDS	get new real stack
031D	34				DES	adjust it for new psuedo stack
031E	34				DES	
031F	7E	020E	R		JMP \$020E	back to original "S"
					END	

We apologize for any inconvenience this may have caused you.

Sincerely,

Blaise W. Liffick
Senior Editor

This short story provides a humorous but tutorial account of the origins of the Tracer program in Jack Hemenway's homebrew 6800 system. It is reprinted here from the December 1977 issue of BYTE, where it first appeared.

Jack and the Machine Debug

"It has to be done by now. That subroutine can't take much more than a few milliseconds per entry, and there aren't many entries. I'll give it a few more seconds." Jack sat nervously puffing his cigar. "It can't take this long," said Jack, his patience exhausted. He punched the RESET button.

"What do you want now, Jack? Here I am, faithfully running your program, and you interrupt me. Find a mistake in your code?"

"Hardly. You should be done by now. What have you been doing that took so long?"

"Well, when you interrupted me, I think I was executing a load-immediate instruction."

"Where?"

"How should I know? You interrupted me. I'm in the monitor ROM now. I can't keep track of every instruction I execute."

"True, true. It sure would be nice if you could, though."

"Well, I can't. I already assemble your programs for you; you can't expect me to debug them for you too! That's supposed to be your department!"

"I know, computer. How do I figure out where you went wrong?"

"How do I know?"

"Calm yourself or I'll use your parts in my F8."

"Okay, Jack. I'm sorry I lost my head. Anything would be better than inflicting that F8 on us. How about trying a breakpoint?"

"Good idea! Computer, sometimes you amaze me. Try a breakpoint at the subroutine return."

"Shouldn't I reload the program first, Jack?"

"I guess so." Jack waited as computer reloaded the program from its cassettes. "Now, put a software interrupt at 1FC0."

"One SWI inserted (hexadecimal 3F to me). Shall I run the program now?"

"Start." Jack went into the kitchen for a beer. He returned a few minutes later. "Computer! What are you doing? RESET!"

"Now what?"

"I told you to set a breakpoint!"

"I did set a breakpoint; see the 3F at 1FC0. I just haven't executed that instruction yet."

"Why not?"

"I haven't the foggiest idea. I just execute them in the order that you wrote them. Writing programs is supposed to be your contribution to our work."

"Don't get snide. Remove the breakpoint."

"Done."

"Now, put the breakpoint at 1FA2."

"I'll reload the program first, Jack."

"I guess you should, but I hate waiting for those cassettes."

"They're your design, remember. If you want speed, buy me some disks."

"They're on order."

"Great. Now let me load the program the best I can from these archaic, cranky, slow, old..."

"Just do the job without the commentary!"

The cassette in the read drive turned ever so slowly. "I'm ready now, Jack. The breakpoint is set."

"Start the program."

Time passed, a lot of time. Jack stabbed

the RESET button hard enough to push the computer across the desk.

"Gently, Jack! I get your message. You must be putting the breakpoint in the wrong place."

"If I knew where to put the breakpoint, then I probably wouldn't need one. What I need is some way to sprinkle a program with breakpoints and just skip the ones I don't need."

"No can do, Jack. My MIKBUG monitor traps every breakpoint and that is that. You can't skip by one. If you put obstacles in my path, I trip over them. You don't want a bruised computer, do you, Jack?"

"I guess not. What I do want is a better way to debug. There's got to be something more effective than this 'stab in the dark' approach."

"May I make a suggestion, Jack?"

"Now look who's the designer. What words of wisdom have you, O great sage of Motorola?"

"Sarcasm will get you nowhere, except maybe 'stabbed in the dark.' I was going to suggest that you investigate my HALT input. If you put a properly timed signal there, then I'll execute only one instruction at a time. You can run programs so slowly even a human can follow the processing."

"That's an interesting idea. Let me think about it for a while."

"I can hardly stop you, Jack. I don't have hands...yet. You were looking at those robot articles in BYTE, weren't you!"

"Talking is quite enough, computer!"

"I...guess...so."

Jack sat back in his chair and thought. Computer knew better than to interrupt such meditations of his human partner. Computer liked its power continuous.

"No good, computer." Jack rolled his chair to the console again. "Hardware single stepping isn't what I need. I need to be able to read your registers and check memory locations. In short, I need your MIKBUG capabilities to help me debug. With your hardware suggestion I'd still need to know where to stop single stepping. That's no better than breakpointing."

"Not exactly, Jack. If you don't muck up my contents with your debugging stuff, then you *can* resume running again after you stop stepping. You can write reentrant code, can't you, Jack?"

"That's exactly what I'm trying to debug. Thanks a bunch."

"Sorry. I guess we'll both have to live with MIKBUG for a while longer, until you write me a real nice monitor, with asynchronous IO, and disks, and..."

"Get off the disk kick. A debugger is what I need. I want a purely software answer. I need to have MIKBUG-like facilities that I can use wherever I want in a program without upsetting that program. It's got to be reentrant. It's got to know how to break down instructions. It should give me a sort of breakpoint for each instruction executed."

"The program you seek is called a tracer. They're available on big machines, like your partner Grappel's PDP-11. Maybe he can adapt one to your liking."

"And adapt it to your limited faculties."

"His big machine can't even talk! Don't you say I'm limited!"

"Okay, okay, I give up. Anyway, it's bedtime. Good night."

"Yeah," said computer. Jack flipped the power switch, and computer's red eye dimmed.

...

"So what's new?" said computer as its fan began to hum.

"Well, I uh...found...discovered that...noticed, uh..."

"Come on, Jack, out with it!"

"That problem you were having yesterday..."

"I wasn't having any problem yesterday! It was your code that was a problem. I just read 'em; I don't write 'em!"

"I know. But you should have warned me that I was pushing one more item onto the stack than I was popping off. When you executed the subroutine return, you got a byte of data confused with the real return address."

"I did not confuse anything! I did exactly, I repeat, exactly, what you asked for. You said PSH, I pushed! You said PUL, I pulled a byte off the stack. You said RTS, and I took the top of the stack as a return address. I may have bugs in the program, but the programmer's got bats in his belfry! If you can't count the number of bytes you put on the stack, you might think of going back to philosophy!"

"Cool it!"

"I might say..."

"Cool it!"

Jack glared at the console, and computer's red eye stared back. "I'm sorry, Jack."

"I guess it really is my fault, computer."

"Friends?"

"Friends."

"Going to get a tracer written?"

"Yep."

"Can I assemble it? I'll do a very careful job."

"I'm sure you will, computer. I'm sure you will."

. . . .

"Computer, let's try to work this breakpoint thing out."

"Glad to help, Jack."

"Fine. Now, we need a program which doesn't change any register or condition code or memory location in the target program. . . the one I need to debug."

"It's got to be reentrant. Right, Jack?"

"It should print the contents of all your registers, the address of the present instruction, and the instruction code. Something like the MIKBUG format should do."

"That's a problem. How do I do all that printing without messing up the registers?"

"Come on, computer. . . that's easy. You save all the registers before printing and then restore them when you're done."

"Like the MIKBUG software interrupt does, on the stack! You know, sometimes you're pretty smart, Jack."

"Except we can't do it that way."

"Why?"

"Because MIKBUG won't let me change the address of the software interrupt handler program. It's in ROM, unfortunately. We'll need another way."

"Jack, isn't this breakpoint thing sort of like a subroutine? I mean, it's, say, 'called' from the target program. . . does some stuff like printing. . . and then returns to the target program."

"I guess we have to do it that way. We'll put a subroutine call (JSR) at the address where the trace is to begin. It will call the trace program, which will be written as a subroutine. The subroutine will first have to save all the registers, then print my debugging info. It can then restore the registers

and return. Thanks for the idea, computer."

"Don't thank me yet; it won't work. If I insert a 3 byte subroutine jump into the target program, then I've destroyed three bytes of your code. Then, when I return from the subroutine, I return three bytes further into the target program, not where I started."

Jack thought a bit and puffed his cigar.

"Jack! That cigar smoke is getting in my cassettes! How can you humans stand all that stuff? Do computers get cancer of the integrated circuit or something?"

"Relax, my automated friend. You're quite safe. I just figured out how to work the tracing."

"I'm all ears."

"I'm surprised you can stop talking long enough to listen. Anyway, I can overcome your objections by careful programming. Before inserting the subroutine jump, you'll save the three bytes you're replacing. You can put them back before you return."

"But, Jack, I still return to the wrong place!"

"Hold it a minute! I can fix up the return address on your stack to back it up three bytes. Then you'll return to the code you've replaced and restored. That'll be a breakpoint that I can really use."

"Glad to help you. But, Jack, you still have to know where to breakpoint. We're scarcely better off than we were with MIKBUG. True, the program can now continue after your breakpoint. Is that all you wanted?"

"It's enough for right now, but we'll probably extend it later. Please assemble this code." Jack placed a cassette in the drive and pressed PLAY. Jack smiled. "It's the only sure way to keep it quiet."

. . . .

"Computer, I want to extend Bob's breakpoint."

"It was only a matter of time. I suppose you want a full trace now."

"Right. It isn't that much more. All a trace is is a moving breakpoint."

"If you can't figure out where you want your breakpoint, then you make me push it around through your stuff. Why is it that I always have to bail you out of your problems?"

"That's what I built you for, remember?"

"Calm down, Jack. I was only kidding."

"I didn't build your sense of humor, that's for sure! Anyway, here's how you'll trace a program. Start with a breakpoint. You handle it in the usual way, except that before you return you put a new breakpoint where the next instruction will be. Effectively, this breakpoints every instruction!"

"Some things are easy to state in words but hard to code. How do I figure out where my next instruction is? I have instructions of different lengths in my op code set. I might jump or branch. . ."

"Computer, remember the 'Thompson Lister' program on page 99 of the October 1976 BYTE? It could figure out how long an instruction was by disassembling your code in memory. Well, I'm going to give you a version of that algorithm so that you can find the next op code. It'll also help you format the instruction printout for my ease in reading."

"Fine. . .if you think you're up to it. Besides, I remember that the 'Thompson Lister' couldn't catch invalid instructions. Sometimes you stick data into a subroutine return address and force me into the middle of nowhere!"

"I remember that incident well enough. I'll add a table of invalid op codes so that you can call me names when you hit one."

"This I like."

"I thought you would. Now, think you can trace?"

Computer sat with lights quivering. "I've got problems, Jack. You've given me a way to find the next instruction in most cases, but what about jumps or branches? Knowing the length of the instruction is no help."

"True. I guess we'll need a set of special cases."

"Oh boy. Here we go."

"It won't be too bad." Jack didn't sound too convinced. "Let's start with the jumps. There are subroutine jumps and unconditional jumps. They can be indexed or extended addressing."

"The subroutine stuff doesn't matter, Jack. For my purposes, a jump is a jump. All I need is the location of the end of the jump."

"Fine. So, we'll have two special cases: extended jumps and indexed jumps. The extended jumps are easy; the second and

third byte of the instruction are the address you require to set your new breakpoint."

"Done."

"The indexed jumps need the contents of the index register from the target program, but you have saved that! You have the offset in the second byte of the instruction! Do a simple addition and you have the new breakpoint address!"

"It's simple if you give me a 16 bit addition program."

"Surely. Now for subroutine returns. You can get the return address from the stack. You've saved the target program stack pointer, so you can get the top of the target stack for your new breakpoint. That's special case 3."

"But what about all the branches?"

"That will take a bit of work. Let's work on the unconditional branches first; they're simpler. You do know where the target program is because you've got its program counter saved. You get the offset from the second byte of the instruction. You just add the offset to the program counter."

"What about signs, Jack?"

"Oh, yes. Forgot about that."

"I noticed that."

"All right, computer. You get a gold star! If the offset is negative, you must subtract it from the program counter. I'll give you a 16 bit subtract too."

"All that for just unconditional branches! I shudder to think what the conditional branches will need."

"Not too much more. We just have to decide whether the branch will be executed or not. If not, then the branch is just another 2 byte instruction. If it is to be executed, then it is equivalent, for your purposes, to an unconditional branch. You've already got code to handle each case."

"Yeah, but how do I know if the branch is to be executed? ESP?"

"Nothing but good, clever programming is needed here. You have the condition codes from the target program saved away. You have the op code, the type of branch. All it takes is a little trick. You'll copy the branch into a spot in the trace code and set the condition codes from your save area. Then, if the branch falls through, you know to treat it as a normal 2 byte instruction. The branch will tell you when to use your branch code. Simple, huh?"

"Self-modifying code. . .very poor form, Jack!"

"Can you do it better?"

"No."

"Then stop complaining. It's effective; it works. Don't knock it."

"At least it will have your name on it and not mine. Any more special cases?"

"A few. We've got to take care of the interrupt instructions RTI and WAI and SWI. Why anybody would try to trace a program with interrupts going off is beyond me, but we'd better be complete. They won't be hard to handle."

"Thank God!"

"Since when did you get religious? Anyhow, the RTI is just like the subroutine return; just the return address is deeper on the stack."

"That was relatively painless. I can figure out the SWI code myself. I know the software interrupt will get a handler address from its vector, which, since I have MIKBUG, is in ROM. My new breakpoint goes at the address found in the vector."

"Very good, computer. Now, the WAI is a bit of a problem. You can't know whether the interrupt that will get you out of wait state will be an IRQ or an NMI. They have different vectors. We'll just have to pick one and warn the user of my tracer that the other type of interrupt causes problems."

"The IRQ is used more often, so I guess I'll get my new address from the IRQ vector."

"I guess that's a good choice."

"Done with special cases, Jack?"

"I think so. Here, I'll load this program and you try to trace it."

Computer began to trace. Jack smiled as the printout overflowed down the printer. Suddenly, the printing stopped. Jack punched RESET.

"I was going good there, wasn't I, Jack?"

"Yeah, but why did you stop?"

"You had this call to MIKBUG in the target program. I traced the next instruction and put my breakpoint out, but then everything fell apart."

"Of course, of course! You can't put breakpoints into ROM! You can try to store anything you want, the data won't change! When you breakpoint, check that your breakpoint is going in. If not, quit before you get lost in thought."

"Now you tell me!"

"Better late than never. Now let's see, we can't trace through ROM or nonexistent memory and we can't tolerate nonmasked interrupts at all, or IRQs unless we were in a wait for interrupt state. Can you think of any other places we'd have trouble?"

"Well, if you hit my RESET then I'll have trouble. I might not have fixed up my breakpoint yet."

"Right. Tell you what: every time you fix up the code after having traced an instruction, wait for me to hit a key on the console. This will let me stop tracing cleanly."

"Glad to oblige. Now, your favorite trick of modifying instructions could cause problems. If an instruction tries to modify the instruction I've tried to breakpoint, well, kaboom!!!"

"Very graphic."

"You're buying me some graphics equipment?"

"No, my eager processor. Perhaps a muzzle. . ."

"Okay. Beware of tracing programs which use modifying instructions. You shouldn't write them that way anyhow."

"Computer, try tracing this now."

The stream of printout began again, with Jack periodically tapping the carriage return key. "Wait a minute, wait a minute! Computer, you're getting some of these branches screwed up."

"I'm just doing what you said to do."

"Well maybe I was wrong."

"Please publish that last comment, Jack! I want that admission in writing!"

"Okay. Now, what's the problem? Why do some branches trace properly and others don't?" Jack poured over the printout while computer hummed contentedly.

"Bob! Come here and look at this!" (Enter Bob, who really was there all the time, but didn't say much.) Bob scanned the trace listing.

"You always get forward branches right. That must be a clue. What is it about backward branches? You get some of them right." Bob thought some more.

"Oh, sure!" Bob jumped to the console again, papers falling to the floor. "If you branch backwards less than three bytes, then your new breakpoint overlaps the present instruction!"

"Fine, Bob. Now what are we to do

about that? My breakpoint has to be three bytes long."

"Yes, but this problem only happens on backwards branches. A branch doesn't change anything in the target program except the program counter. In fact, it needn't be executed at all. We just change the return address from the trace routine to get back to the right place in the target program! We return to the breakpoint call, not the branch! It's easy."

"Fine, Bob. Can I rest now? It's been a long time since I had some time to myself. All work and no play makes Jack's computer dull."

.

"Computer!"

"What is it, Jack? I was just reading that new language you guys have been working on, STRUBAL. Bob wants me to compile it for him. It looks like a big project."

"Well, right now I want you to help me extend our debugger."

"You never give up, do you, Jack?"

"With such an able assistant, why should I?"

"That's hitting below the belt."

"You don't have a belt, computer."

"I forgot," said computer sheepishly. "What now?"

"Your tracing is very helpful, but I'd like to be able to fix the errors that I find without reloading the program and retracing my steps."

"Would you say 'our steps'?"

"If you insist."

"I do."

"Okay. We don't want to retrace our steps. We need more of MIKBUG's capabilities in the debugger. I want to be able to change the register contents in the target program."

"After I spend so much effort saving the contents?"

"Yes. If I find that a register has the wrong thing in it, then I'll want to correct the register before you go on to the next instruction."

"Well, that's no big deal. I just change my stored value for that register. Then, when I return to the target program, the register will have what you want in it. How will you tell me which register to change?"

"I thought a lot about that, and I think I will use the console input that now tells you

to go on. From now on, if I type a carriage return, then go to the next instruction. If I type a capital A, then I want to change your A register. If I type a capital B, then I want to change your B register. Similarly, X and S indicate your index and stack registers. Just after the input you can wait for me to type in the new value I want in that register."

"I suppose I keep letting you change registers until you get around to a carriage return?"

"Right, and, if I type something that doesn't correspond to a register, just skip it. Prompt me for another input."

"Yes sir, boss. Let me anticipate your next request. You want to be able to change memory locations, like MIKBUG does."

"Right again! We'll indicate that with a capital M. I'll enter the address. You give me the present contents and then let me type my desired value for that location."

"Done. I'm going to add a feature that might be useful. I'll automatically convert lower case letters to upper case. Then you won't have to worry about case shifting on that fancy console."

"That's a good idea. Thanks."

"Glad to help. At least it will keep the swearing down when you forget to shift."

"Yes."

"Jack, I've got a question."

"What?"

"If you can change registers and memory at will, can't you get me into situations where I can't continue a trace? Especially if you muck around with the stack."

"I guess that's true, but let the user beware. I don't expect you to protect against every stupidity that a programmer may come up with. All the legitimate cases I can think of will work correctly. After all, the trace program is only about one kilobyte."

"I'm glad you said that and not me."

"Computer, we understand each other."

"Yeah, Jack. Now can I go back to reading STRUBAL?"

"I suppose so."

"Jack, would you put a clean cassette in drive 1? I think I may be needing it."

"Sometimes I wonder who works for whom," muttered Jack as he reached for the bulk eraser. He dropped the cassette into the drive. It began to slowly and in-

exorably turn.

. . . .

"Computer, load the tracer program, please."

"You want to change it *again*!"

"Don't get steamed up. I just want to run an example to test out the tracer."

"What target program should I load?"

"You don't need one."

"Come on, Jack, be serious. Of course I need a target program. You don't expect me to trace memory garbage. You don't mean that, do you, Jack?"

"You've already loaded a program; let's trace that."

"Trace the tracer. Clever! That will really show that tracing doesn't upset the target program. Okay, I'm ready."

"Go."

"What address in the program do you want to start at?"

"How about 212 hexadecimal?"

"212 it is. Here are your registers: index, condition code, B, A, and stack pointer. The instruction is a CLR B, hexadecimal 5F. What would you like?"

"Continue trace." Computer traced the next instruction. Jack typed a carriage return and computer traced again. Again Jack hit the return and computer traced. Jack hit yet another carriage return. Computer traced the instruction at 219.

"Why don't you show off some of your register change stuff? You're at a compare A with 8C immediate instruction; why not make A equal to 8C?"

"Fine. Do it."

"Done. What now?"

"Continue tracing."

"The tracing tracer traces, and having traced, moves on."

"Can the poetry and just trace the program, if you don't mind."

Computer traced the next ten instructions without comment. "Let's show some of the other debug stuff."

"Okay. Change the B register to FF."

"Done."

"Change the index register to 1234."

"Roger."

"Change the condition codes in the target program to D1."

"That's cute, Jack. What does it mean?"

"Just do it."

"All right. How about a memory change?

I've got lots of memory that isn't being used right now."

"Fine. Look at location 500."

"It's got 22 in it now."

"Make that 44, computer."

"Your wish is my command."

"Continue the trace."

"I'm at 10B now. It's a jump to MIKBUG."

Jack hit a carriage return.

"Got to stop here, Jack. I can't trace ROM. Try a new address?"

"No, I think that will make a sufficient example." Jack turned and walked toward the kitchen. He almost imagined that he heard a sigh from the workshop. He ignored it.

And when Tracer was done, Jack's computer sent his printer the following listing of tracer tracing tracer, ultimate confirmation of the program's operation. In this listing, the lines which are blank except for single colons illustrate inputs of carriage returns to cause the program to proceed with tracing the next instruction.

```
ENTER START-TRACE ADDRESS: 0212
X CC B A SP-ADDRESS INSTRUCTION
:0212 D0 02 8C A042 0212 5F
:
:0212 D4 00 8C A042 0213 FE 0173
:
:0216 D0 00 8C A042 0216 A6 00
:
:0216 D8 00 A6 A042 0218 08
:
:0217 D8 00 A6 A042 0219 81 8C
:A 8C
:
:0217 D4 00 8C A042 021B 27 1C
:
:0217 D4 00 8C A042 0239 5C
:
:0217 D0 01 8C A042 023A 5C
:
:0217 D0 02 8C A042 023B 5C
:
:0217 D0 03 8C A042 023C F7 0172
:
:0217 D0 03 8C A042 023F 5A
:
:0217 D0 02 8C A042 0240 27 09
:
:0217 D0 02 8C A042 0242 5A
:
:0217 D0 01 8C A042 0243 27 03
:
:0217 D0 01 8C A042 0245 BD 010B
:B FF
:X 1234
:C D1
```

Tracer

Program Notes

This program was written as an aid to debugging machine language programs on Motorola 6800 microprocessor systems and was designed as an extension to MIKBUG (Motorola's monitor ROM), which is rather weak in debugging facilities. It provides program tracing and register modification functions, extends the MIKBUG memory-change functions and is capable of detecting illegal instructions or other bad code. Consisting of less than 1 K bytes, it is a small package and can be loaded anywhere in memory. It cannot be put in ROM in its present form, however, since instruction modification is used in one spot. It can be used in systems without MIKBUG if IO routines are provided to replace those present in MIKBUG. The IO subroutine calls are made through jumps at the beginning of the program (see listing 1, lines 30 to 37). This facilitates patching the calls to suit other system monitors. Eight routines are used:

BADDR accepts four hex characters from the input device and returns the binary value in the index register.

BYTE accepts two hex characters and returns the binary value in the A register.

INEEE accepts a single ASCII character and returns it in the A register.

OUT2H outputs two hex digits (one byte) to the console device. The byte is pointed to by the index register.

OUT2HS is equivalent to **OUT2H**, except that a space is output after the two hex characters.

OUT4HS outputs four hex digits (2 bytes) pointed to by the index register, followed by a space.

OUTS outputs a single space to the console.

PDATA outputs a string of ASCII codes

to the console until an end of string code (hexadecimal 04) is encountered. The pointer to the string is in the index register.

The stack must be moved to the appropriate address in the system. This patch appears at location **INITER** (line 68 of listing 1).

The program shown here was assembled on a 6800 system using Jack Hemenway's relocating assembler. All absolute addresses are in the extended (2 byte) form, despite the fact that the assembly starts at address zero. To relocate the program to any other address, add the starting address to every memory reference flagged as relocatable by the letter "R" in the listing just to the left of the label field. Note that Jack's assembler does not list the code for FCC assembler pseudo operation. This operation produces ASCII strings with one byte per character in the string. To enter the program from the listing, users must supply the hexadecimal equivalent of the ASCII strings, as found in table 2 on page 21, identified by their address in the listing.

Initialization And Outputs

To run the program, load it in memory (with relocation if necessary) and begin execution (either at the top or at **INITER**). It will then prompt the user for a starting address. This should be the location in memory where one wants to begin debugging. This location should be four hexadecimal digits and should be the first byte of an instruction (if this is not the case, strange results will occur). When the address is entered, the program will print a header line labelling the fields in its diagnostic output. Then it will print the target programs register contents (Index register, condition

codes, B accumulator, A accumulator, stack pointer), the address, and the instruction at that address (one to three bytes). All printout is in hexadecimal form. Remember that these values are true in the target program, not necessarily in the Tracer. After this line, Tracer will print a colon (:) as a prompt and wait for user input.

Inputs

The legal inputs are: letters A, B, C, M, S, X in either upper or lower case, and a carriage return. All other inputs will be ignored and another prompt will be issued. The letters refer to the registers in the Motorola 6800 microprocessor and a carriage return causes Tracer to move on to the next instruction to be executed in the target program though not necessarily the next sequential location. One instruction, that at the address entered, is executed in the target program and the registers are displayed again. The other inputs allow the user to modify the register contents or to alter memory locations in the target program. A, B, and C follow the same syntax. After the letter is input, Tracer prints a space and waits for two hex digits. This value replaces the previous value in the register and a new prompt is issued (the letter C refers to the condition code register). S and X are similar, except that the new value consists of 4 hex digits. The S refers to the stack pointer. M operates like the Motorola MIKBUG memory change function: After the M, Tracer prints a space and waits for an address. After the address Tracer prints a space, the present value of the byte at that address, and a space, then waits for the user to type in two hex digits to form the new value at that address. The lettered commands may be entered in any order and any number of changes can be made at one time. Tracer will not leave the current instruction until it senses a carriage return.

Restrictions

Since Tracer is using a software breakpoint, there are some restrictions to its use. Read only memory (ROM, EROM) cannot be traced. Tracer checks at each step to insure that the next address to be traced is in user programmable memory. If it finds that it cannot trace the next step, it will prompt the user for a new starting address as an indi-

cation that it cannot trace further. This also occurs if the program runs out of memory.

Leaving Tracer

Tracer is an infinite loop. It will run until one stops it by a restart or other interrupt. Restarts should be performed after a prompt for user input. Restarting at other times can leave garbage in random locations in memory.

Interrupts

Nonmaskable interrupts (NMI) cannot be handled by Tracer at all. In fact, it would be rather difficult to trace any interrupt process. Interrupt instructions (WAI, RTI) can be traced, so long as no actual interrupts are occurring.

Self Modifying Code and Undefined Op Codes

Undefined instructions will be flagged as such and Tracer will quit with a prompt for a new address. Some tricky uses of one instruction modifying another will not trace properly. One word of caution; because of this program's extreme flexibility, the user should be able to find many ways of getting into trouble. This is the price one pays for such versatility.

Extension Possibilities

Tracer is readily expandable to include more and fancier capabilities. Through the use of a mnemonic table accessed through the hexadecimal instruction byte, Tracer could give the instruction as a mnemonic rather than as a hexadecimal value or, it could be given hexadecimal arithmetic capability for address calculations during debugging. There are other facilities that could be nice in some conditions; if Tracer was waiting at a prompt, it could be left through a restart and re-entered at label BB1. Thus, one could use other programs to help in debugging while Tracer is retained. In extending or modifying Tracer, remember an important point: the debugger shouldn't become so complex that it becomes a source of trouble rather than an aid in rooting out troublesome bugs.

Listing 1: Tracer assembly and source listing.

```

0001          *          TTF  RRR  A  CCC  EEE  RRR
0002          *          T   R  A  A  C   E   R  R
0003          *          T   RRR AAA  C   EE  RRR
0004          *          T   RR  A  A  C   E   RR
0005          *          T   R  R  A  A  CCC  EEE  R  R  T.M.
0006          *
0007          *
0008          *
0009 0000 0000  N          NAM  TRACER
0010 0000 7E 007A R          JMP  INITER          START VECTOR
0011          *
0012          *
0013          *  A TRACE PROGRAM FOR THE MOTOROLA 6800
0014          *  MICROPROCESSOR.
0015          *
0016          *  COPYRIGHT © 1977 BY ROBERT D. GRAPPEL
0017          *  LEXINGTON MASS. AND JACK E. HEMENWAY
0018          *  BOSTON MASS. ALL RIGHTS RESERVED
0019          *
0020          *  TRACE FAILS AT:
0021          *  1.  ILLEGAL INSTRUCTIONS
0022          *  2.  RESTARTS
0023          *  3.  NMI INTERRUPTS
0024          *  4.  ROM OR UNIMPLEMENTED MEMORY FOUND
0025          *  5.  INSTRUCTION MODIFYING NEXT INSTRUCTION
0026          *
0027          *  USES FOLLOWING MIKBUG LOCATIONS
0028          *  (MIKBUG IS TRADEMARK OF MOTOROLA, INC.)
0029          *
0030 0003 7E E047  BADDR  JMP  $E047
0031 0006 7E E055  BYTE   JMP  $E055
0032 0009 7E E1AC  INEEE  JMP  $E1AC
0033 000C 7E E0BF  OUT2H  JMP  $E0BF
0034 000F 7E E0CA  OUT2HS JMP  $E0CA
0035 0012 7E E0C8  OUT4HS JMP  $E0C8
0036 0015 7E E0CC  OUTS   JMP  $E0CC
0037 0018 7E E07E  PDATA  JMP  $E07E
0038          *
0039 001B A042  STACK  EQU  $A042
0040          *
0041          *
0042 001B 0D0A  TRACEP  FDB  $0D0A          TRACE LINE PROMPT
0043 001D 20          FCC  '  X  CC  B  A  SP-ADDRESS  '
0044 003B 49          FCC  'INSTRUCTION'
0045 0046 2004          FDB  $2004
0046          *
0047 0048 0D0A  CRLF   FDB  $0D0A          CR,LF,COLON
0048 004A 3A04          FDB  $3A04
0049          *
0050 004C 0D0A  PRMPT  FDB  $0D0A          INITIALIZER PROMPT
0051 004E 45          FCC  'ENTER START-TRACE ADDRESS:'
0052 0068 2004          FDB  $2004
0053          *
0054 006A BD      BPNTC  FCB  $BD          'JSR BPHAND '
0055 006B 0097  R      FDB  BPHAND
0056          *
0057 006D 0001  LEN     RMB 1          LENGTH OF INSTRUCTION
0058 006E 0002  PROGC   RMB 2          PROGRAM COUNTER
0059 0070 0002  STACKP  RMB 2          STACK POINTER

```

0060	0072	0002	XREG	RMB 2	X-REGISTER
0061	0074	0001	CCODE	RMB 1	CONDITION CODES
0062	0075	0001	BREG	RMB 1	B-REGISTER
0063	0076	0001	AREG	RMB 1	A-REGISTER
0064	0077	0003	STORE	RMB 3	
0065			*		
0066			* INITIALIZER SECTION		
0067			*		
0068	007A	8E A042	INITER	LDS #STACK	INIT. STACK
0069	007D	CE 004C R		LDX #PRMPT	
0070	0080	BD 0018 R		JSR PDATA	PRINT PROMPT
0071	0083	BD 0003 R		JSR BADDR	GET STARTING ADDRESS
0072	0086	FF 0072 R		SIX XREG	
0073			*		
0074			* SET BREAKPOINT (JSR BPHAND)		
0075			*		
0076	0089	CE 001B R		LDX #TRACEP	PRINT HEADER
0077	008C	BD 0018 R		JSR PDATA	
0078	008F	FE 0072 R		LDX XREG	
0079	0092	BD 02CB R		JSR SETBPT	
0080	0095	6E 00		JMP 0,X	BEGIN TRACING
0082			* BREAKPOINT HANDLER SECTION		
0083			*		
0084	0097	36	BPHAND	PSH A	SAVE MACHINE STATUS
0085	0098	07		TPA	
0086	0099	B7 0074 R		STA A CCODE	(CONDITION CODES)
0087	009C	FF 0072 R		SIX XREG	
0088	009F	32		PUL A	
0089	00A0	B7 0076 R		STA A AREG	
0090	00A3	F7 0075 R		STA B BREG	
0091	00A6	BF 0070 R		STS STACKP	
0092	00A9	CE 0048 R		LDX #CRLF	LINEFEED
0093	00AC	BD 0018 R		JSR PDATA	
0094	00AF	CE 0072 R		LDX #XREG	
0095	00B2	BD 0012 R		JSR OUT4HS	OUTPUT X-REGISTER
0096	00B5	BD 000F R		JSR OUT2HS	OUTPUT C-CODE
0097	00B8	BD 000F R		JSR OUT2HS	OUTPUT B-REGISTER
0098	00BB	BD 000F R		JSR OUT2HS	OUTPUT A-REGISTER
0099	00BE	FE 0070 R		LDX STACKP	GET REAL STACK POINTER
0100	00C1	08		INX	
0101	00C2	08		INX	
0102	00C3	FF 0070 R		SIX STACKP	
0103	00C6	CE 0070 R		LDX #STACKP	
0104	00C9	BD 0012 R		JSR OUT4HS	OUTPUT STACK POINTER
0105	00CC	BD 0015 R		JSR OUTS	SPACES
0106	00CF	BD 0015 R		JSR OUTS	
0107	00D2	BD 0015 R		JSR OUTS	
0108	00D5	30		TSX	
0109	00D6	EE 00		LDX 0,X	
0110	00D8	09		DEX	BACK UP RETURN ADDRESS
0111	00D9	09		DEX	BY 3 BYTES
0112	00DA	09		DEX	
0113	00DB	FF 006E R		SIX PROGC	
0114	00DE	B6 006E R		LDA A PROGC	
0115	00E1	F6 006F R		LDA B PROGC+1	
0116	00E4	30		TSX	
0117	00E5	E7 01		STA B 1,X	
0118	00E7	A7 00		STA A 0,X	
0119	00E9	FE 006E R		LDX PROGC	BYTE 3 OF INSTRUCTION

0120	00EC	B6	0079	R	LDA A STORE+2	
0121	00EF	A7	02		STA A 2,X	
0122	00F1	B6	0078	R	LDA A STORE+1	BYTE 2
0123	00F4	A7	01		STA A 1,X	
0124	00F6	B6	0077	R	LDA A STORE	BYTE 1
0125	00F9	A7	00		STA A 0,X	
0126					*	
0127					* NOW DECODE INSTRUCTION	
0128					*	
0129	00FB	CE	006E	R	LDX #PROGC	
0130	00FE	BD	0012	R	JSR OUT4HS	OUTPUT INSTRUCTION ADDRESS
0131	0101	BD	0015	R	JSR OUTS	SPACE
0132	0104	BD	0015	R	JSR OUTS	SPACE
0133	0107	FE	006E	R	LDX PROGC	
0134	010A	BD	000F	R	JSR OUT2HS	OUTPUT INSTRUCTION BYTE
0135					*	
0136					* NOW COMPUTE INSTRUCTION LENGTH	
0137					* PARTIAL DISASSEMBLY DONE HERE	
0138					*	
0139	010D	5F			CLR B	LENGTH=1
0140	010E	FE	006E	R	LDX PROGC	
0141	0111	A6	00		LDA A 0,X	GET INSTRUCTION BYTE
0143					*	
0144					* ILLEGAL INSTRUCTION TRAP	
0145					*	
0146	0113	CE	0135	R	LDX #ILTBL	POINT TO BAD CODE TABLE
0147	0116	A1	00		ILL00P CMP A 0,X	TEST MATCH
0148	0118	27	08		BEQ BADCOD	FOUND MATCH?
0149					*	
0150	011A	08			INX	
0151	011B	8C	016F	R	CPX #ILEND	END OF TABLE?
0152	011E	26	F6		BNE ILL00P	NO, KEEP LOOKING
0153					*	
0154	0120	6E	00		JMP 0,X	VALID OPCODE
0155					*	
0156	0122	CE	012B	R	BADCOD LDX #BADPRT	
0157	0125	BD	0018	R	JSR PDATA	OUTPUT MESSAGE
0158	0128	7E	007A	R	JMP INITER	QUIT
0159					*	
0160	012B	55			BADPRT FCC 'UNDEFINED'	
0161	0134	04			FCB \$04	
0162					*	
0163	0135	0003			ILTBL FDB \$0003	UNDEFINED OPCODES
0164	0137	0405			FDB \$0405	FOR M6800
0165	0139	1213			FDB \$1213	
0166	013B	1415			FDB \$1415	
0167	013D	181A			FDB \$181A	
0168	013F	1C1D			FDB \$1C1D	
0169	0141	1E1F			FDB \$1E1F	
0170	0143	2138			FDB \$2138	
0171	0145	3A3C			FDB \$3A3C	
0172	0147	3D41			FDB \$3D41	
0173	0149	4245			FDB \$4245	
0174	014B	4B4E			FDB \$4B4E	
0175	014D	5152			FDB \$5152	
0176	014F	555B			FDB \$555B	
0177	0151	5E61			FDB \$5E61	
0178	0153	6265			FDB \$6265	
0179	0155	6B71			FDB \$6B71	

0180	0157	7275		FDB \$7275	
0181	0159	7B83		FDB \$7B83	
0182	015B	878F		FDB \$878F	
0183	015D	939D		FDB \$939D	
0184	015F	A3B3		FDB \$A3B3	
0185	0161	C3C7		FDB \$C3C7	
0186	0163	CCCC		FDB \$CCCC	
0187	0165	CFD3		FDB \$CFD3	
0188	0167	DCDD		FDB \$DCDD	
0189	0169	E3EC		FDB \$E3EC	
0190	016B	EDF3		FDB \$EDF3	
0191	016D	FCFD		FDB \$FCFD	
0192			*		
0193	016F	FE 006E	R ILEND	LDX PROGC	RESTORE X-REGISTER
0195	0172	08		INX	
0196	0173	81 8C		CMP A #\$8C	CPX?
0197	0175	27 1C		BEQ B3	3-BYTES
0198			*		
0199	0177	81 8E		CMP A #\$8E	LDS?
0200	0179	27 18		BEQ B3	3-BYTES
0201			*		
0202	017B	81 CE		CMP A #\$CE	LDX?
0203	017D	27 14		BEQ B3	3-BYTES
0204			*		
0205	017F	81 8D		CMP A #\$8D	BSR?
0206	0181	27 11		BEQ B2	2-BYTES
0207			*		
0208	0183	84 F0		AND A #\$F0	
0209	0185	81 20		CMP A #\$20	BRANCH?
0210	0187	27 0B		BEQ B2	2-BYTES
0211			*		
0212	0189	81 60		CMP A #\$60	
0213	018B	25 08		BCS B1	1-BYTE
0214			*		
0215	018D	84 30		AND A #\$30	
0216	018F	81 30		CMP A #\$30	
0217	0191	26 01		BNE B2	2-BYTES
0218			*		
0219	0193	5C	B3	INC B	3-BYTE INSTRUCTION
0220	0194	5C	B2	INC B	2-BYTE INSTRUCTION
0221	0195	5C	B1	INC B	1-BYTE INSTRUCTION
0222	0196	F7 006D	R	STA B LEN	
0223	0199	5A		DEC B	
0224	019A	27 09		BEQ BB1	
0225			*		
0226	019C	5A		DEC B	
0227	019D	27 03		BEQ BB2	
0228			*		
0229	019F	BD 000C	R BB3	JSR OUT2H	3-BYTE INSTRUCTION OUTPUT
0230	01A2	BD 000C	R BB2	JSR OUT2H	2-BYTE
0231	01A5	CE 0048	R BB1	LDX #CRLF	
0232	01A8	BD 0018	R	JSR PDATA	LINEFEED
0233			*		
0235			*	STATUS CHANGE SECTION	
0236			*		
0237	01AB	BD 0009	R	JSR INEE	WAIT FOR KEYPRESS
0238	01AE	84 DF		AND A #\$DF	FORCE UPPER CASE
0239	01B0	81 0D		CMP A #\$0D	CR?
0240	01B2	27 6B		BEQ DECD	IF SO, CONTINUE TRACE

0241				*		
0242	01B4	81	41		CMP A #\$41	"A"?
0243	01B6	26	0B		BNE NEXT1	NO
0244				*		
0245				* CHANGE A-REGISTER CONTENTS		
0246				*		
0247	01B8	BD	0015 R		JSR OUTS	SPACE
0248	01BB	BD	0006 R		JSR BYTE	GET DATA
0249	01BE	B7	0076 R		STA A AREG	STORE IT
0250	01C1	20	E2		BRA BB1	GET NEW KEY
0251				*		
0252	01C3	81	42	NEXT1	CMP A #\$42	"B"?
0253	01C5	26	0B		BNE NEXT2	NO
0254				*		
0255				* CHANGE B-REGISTER CONTENTS		
0256				*		
0257	01C7	BD	0015 R		JSR OUTS	SPACE
0258	01CA	BD	0006 R		JSR BYTE	GET DATA
0259	01CD	B7	0075 R		STA A BREG	STORE IT
0260	01D0	20	D3		BRA BB1	GET NEW KEY
0261				*		
0262	01D2	81	43	NEXT2	CMP A #\$43	"C"?
0263	01D4	26	0B		BNE NEXT3	NO
0264				*		
0265				* CHANGE CONDITION CODES		
0266				*		
0267	01D6	BD	0015 R		JSR OUTS	SPACE
0268	01D9	BD	0006 R		JSR BYTE	GET DATA
0269	01DC	B7	0074 R		STA A CCODE	STORE IT
0270	01DF	20	C4		BRA BB1	GET NEW KEY
0271				*		
0272	01E1	81	4D	NEXT3	CMP A #\$4D	"M"?
0273	01E3	26	1C		BNE NEXT4	NO
0274				*		
0275				* CHANGE MEMORY LOCATION		
0276				*		
0277	01E5	BD	0015 R		JSR OUTS	SPACE
0278	01E8	BD	0003 R		JSR BADDR	GET MEMORY ADDRESS
0279	01EB	FF	0077 R		STX STORE	
0280	01EE	BD	0015 R		JSR OUTS	SPACE
0281	01F1	FE	0077 R		LDX STORE	
0282	01F4	BD	000F R		JSR OUT2HS	PRINT CONTENTS
0283	01F7	BD	0006 R		JSR BYTE	GET DATA BYTE
0284	01FA	FE	0077 R		LDX STORE	
0285	01FD	A7	00		STA A 0,X	STORE IT AT ADDRESS
0286	01FF	20	A4		BRA BB1	GET NEW KEY
0287				*		
0288	0201	81	53	NEXT4	CMP A #\$53	"S"?
0289	0203	26	0B		BNE NEXT5	NO
0290				*		
0291				* CHANGE STACK POINTER		
0292				*		
0293	0205	BD	0015 R		JSR OUTS	SPACE
0294	0208	BD	0003 R		JSR BADDR	GET NEW VALUE
0295	020B	FF	0070 R		STX STACKP	STORE IT
0296	020E	20	95		BRA BB1	GET NEW KEY
0297				*		
0298	0210	81	58	NEXT5	CMP A #\$58	"X"?
0299	0212	26	91		BNE BB1	IF NOT, GET NEW KEY

0300				
0301			*	
0302			* CHANGE X-REGISTER CONTENTS	
0303	0214	BD	0015	R JSR OUTS SPACE
0304	0217	BD	0003	R JSR BADDR GET NEW VALUE
0305	021A	FF	0072	R STX XREG STORE IT
0306	0 1D	20	86	BRA BBI GET NEW KEY
0307			*	
0308			*	
0310			* DECODE SPECIAL CASES HERE	
0311			* FIND NEXT INSTRUCTION'S ADDRESS	
0312			*	
0313	021F	FE	006E	R DECOD LDX PROGC
0314	0222	A6	00	LDA A 0,X GET INSTRUCTION BYTE
0315			*	
0316			* INDEXED JUMPS HERE	
0317			*	
0318	0224	81	6E	CMP A #\$6E JMP X?
0319	0226	27	2C	BEQ INDEX
0320	0228	81	AD	CMP A #\$AD JSR X?
0321	022A	27	28	BEQ INDEX
0322			*	
0323			* EXTENDED JUMPS HERE	
0324			*	
0325	022C	81	7E	CMP A #\$7E JMP EXT?
0326	022E	27	20	BEQ EXTEND
0327	0230	81	BD	CMP A #\$BD JSR EXT?
0328	0232	27	1C	BEQ EXTEND
0329			*	
0330			* SUBROUTINE HANDLING	
0331			*	
0332	0234	81	8D	CMP A #\$8D BSR?
0333	0236	27	34	BEQ BRNCH1
0334	0238	81	39	CMP A #\$39 RTS?
0335	023A	27	64	BEQ RTSUB
0336			*	
0337			* INTERRUPT INSTRUCTIONS	
0338			*	
0339	023C	81	3B	CMP A #\$3B RTI?
0340	023E	27	71	BEQ RTIZ
0341	0240	81	3F	CMP A #\$3F SWI?
0342	0242	27	63	BEQ SWIZ
0343	0244	81	3E	CMP A #\$3E WAI?
0344	0246	27	64	BEQ WAIZ
0345			*	
0346			* BRANCHES	
0347			*	
0348	0248	84	FO	AND A #\$FO
0349	024A	81	20	CMP A #\$20
0350	024C	27	11	BEQ BRANCH
0351			*	
0352	024E	20	68	BRA NORMAL ALL OTHERS
0353			*	
0354			* EXTENDED JUMPS	
0355			*	
0356	0250	EE	01	EXTEND LDX 1,X GET JUMP ADDRESS
0357	0252	20	77	BRA SETBPT RESET BREAKPOINT
0358			*	

0359				* INDEXED JUMPS		
0360				*		
0361	0254	5F		INDEX	CLR B	
0362	0255	A6	01		LDA A 1,X	GET OFFSET
0363	0257	BB	0073 R		ADD A XREG+1	ADD IN X-REGISTER
0364	025A	F9	0072 R		ADC B XREG	
0365	025D	20	63		BRA UPDATE	RESET BREAKPOINT
0366				*		
0367				* BRANCH INSTRUCTIONS		
0368				*		
0369	025F	A6	00	BRANCH	LDA A 0,X	GET BRANCH TYPE
0370				*		
0371				* CONDITIONAL BRANCH TEST		
0372				*		
0373	0261	B7	0268 R		STA A TEST	INSTRUCTION INSERTED
0374	0264	B6	0074 R		LDA A CCODE	SET CONDITION CODE
0375	0267	06			TAP	
0376	0268	2J	02	TEST	BRA **4	MODIFIED INSTRUCTION
0377	026A	20	4C		BRA NORMAL	NO BRANCH
0378				*		
0379	026C	A6	01	BRNCH1	LDA A 1,X	GET OFFSET
0380	026E	5F			CLR B	
0381	026F	03			INX	ADD 2 TO PROGC
0382	0270	08			INX	
0383	0271	FF	006E R		STX PROGC	
0384	0274	4D			TST A	OFFSET PLUS OR MINUS?
0385	0275	2D	08		BLT BRNCH2	IF MINUS, SUBTRACT
0386				*		
0387				* FORWARD BRANCH		
0388				*		
0389	0277	BB	006F R		ADD A PROGC+1	OTHERWISE ADD
0390	027A	F9	006E R		ADC B PROGC	
0391	027D	20	43		BRA UPDATE	
0392				*		
0393				* BACKWARD BRANCH		
0394				*		
0395	027F	40		BRNCH2	NEG A	
0396	0280	B0	006F R		SUB A PROGC+1	
0397	0283	F2	006E R		SBC B PROGC	
0398	0286	43			COM A	MINUS ACCUMS.
0399	0287	53			COM B	
0400	0288	8B	01		ADD A #1	
0401	028A	C9	00		ADC B #0	
0402				*		
0403	028C	36			PSH A	SAVE "A"
0404	028D	FE	006E R		LDX PROGC	
0405	0290	09			DEX	
0406	0291	09			DEX	
0407	0292	A6	00		LDA A 0,X	REGAIN BRANCH TYPE
0408	0294	81	8D		CMP A #\$8D	CHECK FOR BSR
0409	0296	32			PUL A	RESTORE "A"
0410	0297	27	29		BEQ UPDATE	IF BSR, MUST EXECUTE IT
0411				*		
0412	0299	30		MRET	TSX	MODIFY RETURN POINT
0413	029A	A7	01		STA A 1,X	
0414	029C	E7	00		STA B 0,X	
0415	029E	20	22		BRA UPDATE	FINISH UP
0416				*		

0417				* SUBROUTINE RETURN	
0418				*	
0419	02A0	FE	0070 R	RTSUB	LDX STACKP
0420	02A3	EE	01		LDX 1,X
0421	02A5	20	24		BRA SETBPT
0422				*	
0423				* SOFTWARE INTERRUPT	
0424				*	
0425	02A7	FE	FFFA	SWIZ	LDX \$FFFA
0426	02AA	20	1F		BRA SETBPT
0427				*	
0428				* WAI INSTRUCTION--ASSUME IRQ WILL TERMINATE	
0429				*	
0430	02AC	FE	FFF8	WAIZ	LDX \$FFF8
0431	02AF	20	1A		BRA SETBPT
0432				*	
0433				* RETURN FROM INTERRUPT	
0434				*	
0435	02B1	FE	0070 R	RTIZ	LDX STACKP
0436	02B4	EE	06		LDX 6,X
0437	02B6	20	13		BRA SETBPT
0438				*	
0439				* ALL OTHERS	
0440				*	
0441	02B8	B6	006D R	NORMAL	LDA A LEN
0442	02B8	5F			CLR B
0443	02BC	BB	006F R		ADD A PROGC+1
0444	02BF	F9	006E R		ADC B PROGC
0445				*	
0446	02C2	B7	006F R	UPDATE	STA A PROGC+1
0447	02C5	F7	006E R		STA B PROGC
0448				*	
0450				* RESET BREAKPOINT AND RETURN	
0451				*	
0452	02C8	FE	006E R		LDX PROGC
0453	02CB	A6	00	SETBPT	LDA A 0,X
0454	02CD	B7	0077 R		STA A STORE
0455	02D0	F6	006A R		LDA B BPNTC
0456	02D3	E7	00		STA B 0,X
0457	02D5	A6	00		LDA A 0,X
0458	02D7	11			CBA
0459	02D8	26	2E		BNE QUIT
0460				*	
0461	02DA	A6	01		LDA A 1,X
0462	02DC	B7	0078 R		STA A STORE+1
0463	02DE	F6	006B R		LDA B BPNTC+1
0464	02E2	E7	01		STA B 1,X
0465	02E4	A6	01		LDA A 1,X
0466	02E6	11			CBA
0467	02E7	26	1F		BNE QUIT
0468				*	
0469	02E9	A6	02		LDA A 2,X
0470	02EB	B7	0079 R		STA A STORE+2
0471	02EE	F6	006C R		LDA B BPNTC+2
0472	02F1	E7	02		STA B 2,X
0473	02F3	A6	02		LDA A 2,X
0474	02F5	11			CBA
0475	02F6	26	10		BNE QUIT

```

0476          *
0477 02F8 FE 0072 R      LDX XREG          RESTORE X-REGISTER
0478 02F8 F6 0075 R      LDA B BREG          RESTORE B-REGISTER
0479 02FE B6 0076 R      LDA A AREG
0480 0301 36             PSH A
0481 0302 B6 0074 R      LDA A CCODE
0482 0305 06             TAP                RESTORE CONDITION CODES
0483 0306 32             PUL A                RESTORE A-REGISTER
0484 0307 39             RTS
0485          *
0486          *
0487 0308 7E 007A R QUIT  JMP INITER          ERROR TRAP
0488          *
0489          END

```

Table 1: Sorted symbol table for the above assembly.

AREG	0076 R	BRNCH1	026C R	MRET	0299 R	QUIT	0308 R
B1	0195 R	BRNCH2	027F R	NEXT1	01C3 R	RTIZ	02B1 R
B2	0194 R	BYTE	0006 R	NEXT2	01D2 R	RTSUB	02A0 R
B3	0193 R	CCODE	0074 R	NEXT3	01E1 R	SETBPT	02CB R
BADCOD	0122 R	CRLF	0048 R	NEXT4	0201 R	STACK	A042
BADDR	0003 R	DECOD	021F R	NEXT5	0210 R	STACKP	0070 R
BADPRT	012B R	EXTEND	0250 R	NORMAL	02B8 R	STORE	0077 R
BB1	01A5 R	ILEND	016F R	OUT2H	000C R	SWIZ	02A7 R
BB2	01A2 R	ILLOOP	0116 R	OUT2HS	000F R	TEST	0268 R
BB3	019F R	ILTBL	0135 R	OUT4HS	0012 R	TRACEP	001B R
BPHAND	0097 R	INDEX	0254 R	OUTS	0015 R	TRACER	0000 RN
BPNTC	006A R	INEEE	0009 R	PDATA	0018 R	UPDATE	02C2 R
BRANCH	025F R	INITER	007A R	PRMPT	004C R	WAITZ	02AC R
BREG	0075 R	LEN	006D R	PROGC	006E R	XREG	0072 R

Table 2: Table of hexadecimal data for the character strings of listing 1. Each string is identified by its symbol and address as in listing 1.

Address	Hexadecimal Data for String "TRACEP"								Address	Hexadecimal Data for String "PRMPT"							
001B	0D	0A	20	20	20	58	20	20	004C	0D	0A	45	4E	54	45	52	20
0023	43	43	20	20	42	20	20	41	0054	53	54	41	52	54	2D	54	52
002B	20	20	20	53	50	2D	41	44	005C	41	43	45	20	41	44	44	52
0033	44	52	45	53	53	20	20	20	0064	45	53	53	3A	20	04		
003B	49	4E	53	54	52	55	43	54									
0043	49	4F	4E	20	04												
Address	Hexadecimal Data for String "CRLF"								Address	Hexadecimal Data for String "BADPRT"							
0048	0D	0A	3A	04					012B	55	4E	44	45	46	49	4E	45
									0133	44	04						

Beginning on the next page is a complete machine readable representation of the object code for Tracer, as assembled in the listing found on pages 13 to 21 of this book.

This Tracer representation uses the absolute loader format, in which each bar code frame (one line of bars running from top to bottom of the page) contains a two byte address followed by data which is loaded in ascending order starting at that address.

The object code listing shown below gives the information in hexadecimal form, with one line per Bar Code frame for use as a confirmation copy.

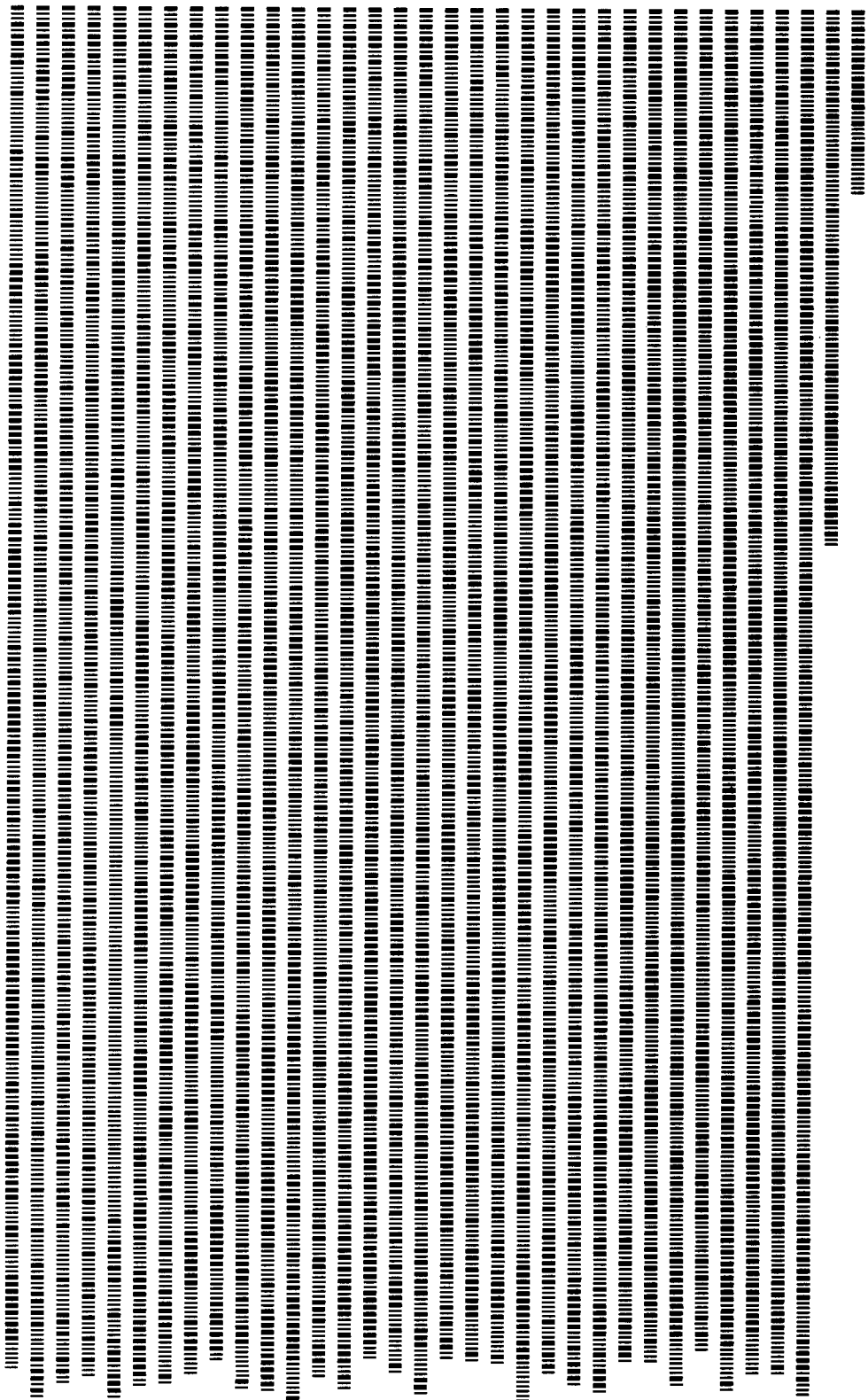
For details on the frame format and absolute loader format used in this and all Paperbyte™ Books, see the Paperbyte™ Publication "Bar Code Loader" by Ken Budnick. This book, the first in the Paperbyte™ series, contains a brief history on bar codes, a general bar code loader algorithm with flow charts and complete program listings for 6800, 6502 and 8080 or Z-80 based systems.

```

0100 7E 01 7A 7E E0 47 7E E0 55 7E E1 AC 7E E0 BF 7E
0110 E0 CA 7E E0 C8 7E E0 CC 7E E0 7E 0D 0A 20 20 20
0120 58 20 20 43 43 20 20 42 20 20 41 20 20 20 53 50
0130 2D 41 44 44 52 45 53 20 20 04 0D 0A 3A 04 0D 0A 45 4E
0140 55 43 54 49 4F 4E 20 04 0D 0A 3A 04 0D 0A 45 4E
0150 54 45 52 20 53 54 41 52 54 2D 54 52 41 43 45 20
0160 41 44 44 52 45 53 53 3A 20 04 BD 01 97 00 00 00
0170 00 00 00 00 00 00 00 00 00 00 8E A0 42 CE 01 4C
0180 BD 01 18 BD 01 03 FF 01 72 CE 01 1B BD 01 18 FE
0190 01 72 BD 03 CB 6E 00 36 07 B7 01 74 FF 01 72 32
01A0 B7 01 76 F7 01 75 BF 01 70 CE 01 48 BD 01 18 CE
01B0 01 72 BD 01 12 BD 01 0F BD 01 0F BD 01 0F FE 01
01C0 70 08 08 FF 01 70 CE 01 70 BD 01 12 BD 01 15 BD
01D0 01 15 BD 01 15 30 EE 00 09 09 09 FF 01 6E B6 01
01E0 6E F6 01 6F 30 E7 01 A7 00 FE 01 6E B6 01 79 A7
01F0 02 B6 01 78 A7 01 B6 01 77 A7 00 CE 01 6E BD 01
0200 12 BD 01 15 BD 01 15 FE 01 6E BD 01 0F 5F FE 01
0210 6E A6 00 CE 02 35 A1 00 27 08 08 8C 02 6F 26 F6
0220 6E 00 CE 02 2B BD 01 18 7E 01 7A 55 4E 44 45 46
0230 49 4E 45 44 04 00 03 04 05 12 13 14 15 18 1A 1C
0240 1D 1E 1F 21 38 3A 3C 3D 41 42 45 4B 4E 51 52 55
0250 5B 5E 61 62 65 6B 71 72 75 7B 83 87 8F 93 9D A3
0260 B3 C3 C7 CC CD CF D3 DC DD E3 EC ED F3 FC FD FE
0270 01 6E 08 81 8C 27 1C 81 8E 27 18 81 CE 27 14 81
0280 8D 27 11 84 F0 81 20 27 0B 81 60 25 08 84 30 81
0290 30 26 01 5C 5C 5C F7 01 6D 5A 27 09 5A 27 03 BD
02A0 01 0C BD 01 0C CE 01 48 BD 01 18 BD 01 09 84 DF
02B0 81 0D 27 6B 81 41 26 0B BD 01 15 BD 01 06 B7 01
02C0 76 20 E2 81 42 26 0B BD 01 15 BD 01 06 B7 01 75
02D0 20 D3 81 43 26 0B BD 01 15 BD 01 06 B7 01 74 20
02E0 C4 81 4D 26 1C BD 01 15 BD 01 03 FF 01 77 BD 01
02F0 15 FE 01 77 BD 01 0F BD 01 06 FE 01 77 A7 00 20
0300 A4 81 53 26 0B BD 01 15 BD 01 03 FF 01 70 20 95
0310 81 58 26 91 BD 01 15 BD 01 03 FF 01 72 20 86 FE
0320 01 6E A6 00 81 6E 27 2C 81 AD 27 28 81 7E 27 20
0330 81 BD 27 1C 81 8D 27 34 81 39 27 64 81 3B 27 71
0340 81 3F 27 63 81 3E 27 64 84 F0 81 20 27 11 20 68
0350 EE 01 20 77 5F A6 01 BB 01 73 F9 01 72 20 63 A6
0360 00 B7 03 68 B6 01 74 06 20 02 20 4C A6 01 5F 08
0370 08 FF 01 6E 4D 2D 08 BB 01 6F F9 01 6E 20 43 40
0380 B0 01 6F F2 01 6E 43 53 8B 01 C9 00 36 FE 01 6E
0390 09 09 A6 00 81 8D 32 27 29 30 A7 01 E7 00 20 22
03A0 FE 01 70 EE 01 20 24 FE FF FA 20 1F FE FF F8 20
03B0 1A FE 01 70 EE 06 20 13 B6 01 6D 5F BB 01 6F F9
03C0 01 6E B7 01 6F F7 01 6E FE 01 6E A6 00 B7 01 77
03D0 F6 01 6A E7 00 A6 00 11 26 2E A6 01 B7 01 78 F6
03E0 01 6B E7 01 A6 01 11 26 1F A6 02 B7 01 79 F6 01
03F0 6C E7 02 A6 02 11 26 10 FE 01 72 F6 01 75 B6 01
0400 76 36 B6 01 74 06 32 39 7E 01 7A

```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	3	4	6	7	9	A	C	D	F	0	2	2	2	2	2	2	2	2	2
0	7	1	A	3	E	6	E	6	E	6	E	7	0	8	E	6	F	7	F	7



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9

A Note About Bar Codes . . .

Bar codes are the newest form of machine readable data representation. They are used in all PAPERBYTE™ software products in BYTE magazine articles and self contained book publications and combine efficiency of space, low cost, and ease of data entry with the need for mass produced machine readable representations of software. Bar codes were originally used for product identification in inventory control and supermarket checkout applications. Today, because of their direct binary representation of data, they are an ideal computer compatible communications medium. In the application of bar codes to software distribution (such as PAPERBYTE books and articles), the use of a simple but reliable optical scanning wand and an appropriate program provides a convenient means for the user to acquire software.

Our intent in making PAPERBYTE software available in bar code form is to provide a method of conveying machine readable information from documentation to the memories and mass storage of a user's system on a one time basis. We suggest that the user of software obtained in this manner should locally record the data on the mass storage devices of his system after the data has been scanned from the printed page. The PAPERBYTE bar code representations provide a standardized means of obtaining the data, but they cannot be compared to the convenience of local mass storage devices such as floppy disks, digital cassettes or audio cassettes. Thus if repeated use of the software obtained from bar code is anticipated, we recommend that the user make a copy on some form of magnetic medium.

Bar Code Loader by Ken Budnik, the first in the PAPERBYTE series of software books, provides a brief history of bar codes, a look at the PAPERBYTE bar code format including flowcharts, a general bar code loader algorithm and well documented programs with complete implementation and checkout procedures for 6800, 6502 and 8080/Z-80 based systems.

Bar Codes Provided by:

Walter Banks

Computer Communications Network Group

University of Waterloo

Waterloo, Ontario, Canada

Tracer: A 6800 Debugging Program

featuring

Single step execution using dynamic break points . . .
Register examination and modification . . .
Memory examination and modification . . .

by

Robert D Grappel and **Jack E Hemenway**

This publication includes

- "Jack And The Machine Debug"
- Tracer Program Notes
- Complete Assembly and Source Listing
- Complete Object Code Listing
- Machine Readable Object Code

0-07-024121-x

